# Introduction to Transactions

# Recap

- A distributed system is a collection of independent computers that appears to its users as a single coherent system.
- To do so, they have to cooperate, that is to *synchronise*
  - Synchronise the clock
  - Synchronise concurrent actions

# Synchronisation

When a server uses multiple threads it can perform several client operations concurrently

Clients share resources via a server

- Synchronisation at server
- Transactions

# Example

- deposit(amount)
  - deposit amount in the account
- withdraw(amount)
  - withdraw amount from the account
- getBalance() → amount
  - return the balance of the account
- setBalance(amount)
  - set the balance of the account to amount

# Synchronisation at Server (1)

- When a server uses multiple threads it can perform several client operations concurrently
- If we allowed *deposit* and *withdraw* to run concurrently we could get inconsistent results
- Objects should be designed for safe concurrent access e.g. in Java use synchronized methods, e.g.
  - *public synchronized void deposit(int amount) throws RemoteException*
- *Atomic operations* are free from interference from concurrent operations in other threads.
- Use any available mutual exclusion mechanism

# Synchronisation at Server (2)

- Clients share resources via a server
  - e.g. some clients update server objects and others access them
- In some applications, clients depend on one another to progress
  - e.g. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released
- It would not be a good idea for a waiting client to poll the server to see whether a resource is yet available
  - it would also be unfair (later clients might get earlier turns)
- Needs **extensive programming** e.g. in Java (tools available)

# More Problems: Failures

- Writes to permanent storage may fail
  - e.g. by writing nothing or a wrong value (write to wrong block is a disaster)
- Servers may crash occasionally.
  - when a crashed server is replaced by a new process its memory is cleared and then it carries out a recovery procedure to get its objects' state
- There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted.
  - recipient can detect corrupt messages (by checksum)
  - forged messages and undetected corrupt messages are disasters

# Transactions

- Some applications require a sequence of client requests to a server to be *atomic* in the sense that:
  - they are free from interference by operations being performed on behalf of other concurrent clients; and
  - either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

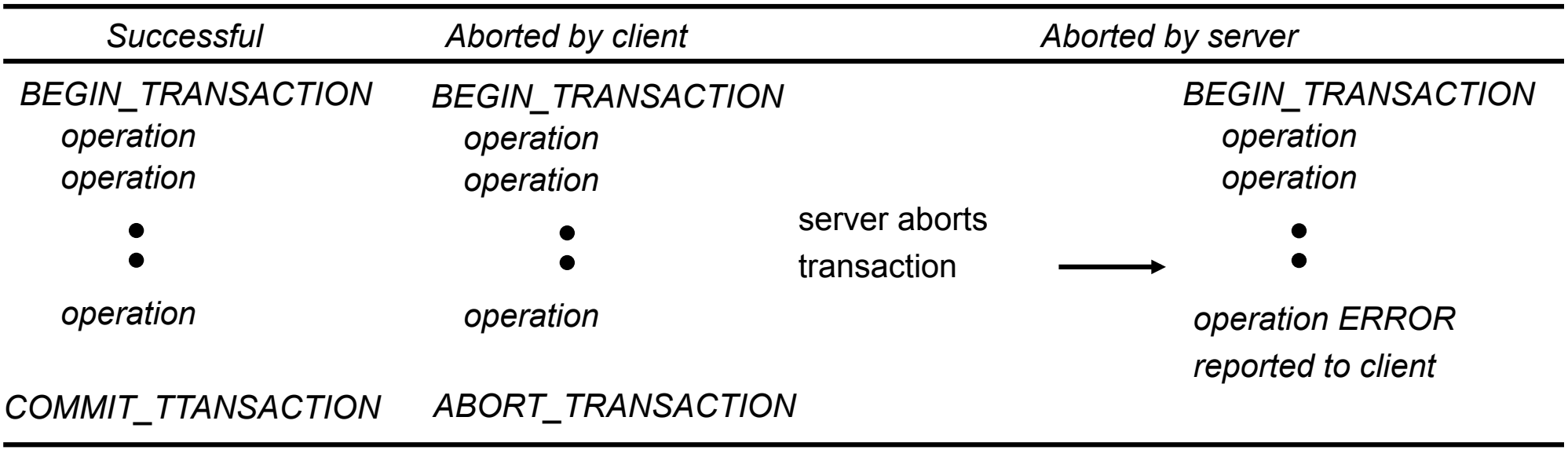
  Such sequences are known as transactions

# Banking Transaction

*Transaction T:*
*a.withdraw(100);*
*b.deposit(100);*
*c.withdraw(200);*
*b.deposit(200);*

- This transaction specifies a sequence of related operations involving bank accounts named *A*, *B* and *C* and referred to as *a, b* and *c* in the program
  - The first two operations transfer $100 from *A* to B
  - The second two operations transfer $200 from *C* to *B*

# Transaction Life Histories




- A transaction is either successful (it commits)
  - all objects are saved in permanent storage
- …or it is aborted by the client or the server
  - make all temporary effects invisible to other transactions

# The Transaction Model

| Primitive | Description |
|---|---|
| BEGIN_TRANSACTION | Make the start of a transaction |
| END_TRANSACTION | Terminate the transaction and try to commit |
| ABORT_TRANSACTION | Kill the transaction and restore the old values |
| READ | Read data from a file, a table, or otherwise |
| WRITE | Write data to a file, a table, or otherwise |

Examples of primitives for transactions.

# Transactions: Not Only Databases!

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

(b)

a) Transaction to reserve three flights "commits".

b) Transaction "aborts" when third flight is unavailable.

# A.C.I.D.

Four key transaction characteristics:

**Atomic**: the transaction is considered to be one thing, even though it may be made of up many different parts.

**Consistent**: "invariants" that held before the transaction must also hold after its successful execution.

**Isolated**: if multiple transactions run at the same time, they must not interfere with each other. To the system, it should look like the two (or more) transactions are executed sequentially (i.e., that they are *serializable*).

**Durable**: Once a transaction commits, any changes are permanent.

# Types of Transactions

- **Flat Transaction**: this is the model that we have looked at so far. Disadvantage: too rigid. Partial results cannot be committed. That is, the "atomic" nature of Flat Transactions can be a downside.
- **Nested Transaction**: a main, parent transaction spawns child sub-transactions to do the real work. Disadvantage: problems result when a sub-transaction commits and then the parent aborts the main transaction. Things get messy.
- **Distributed Transaction**: this is sub-transactions operating on distributed data stores. Disadvantage: complex mechanisms required to lock the distributed data, as well as commit the entire transaction.

# Serialisability

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

| | | |
|---|---|---|
| Schedule 1 | x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3 | Legal |
| Schedule 2 | x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3; | Legal |
| Schedule 3 | x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3; | Illegal |

(d)

a) – c) Three transactions $T_1$, $T_2$, and $T_3$

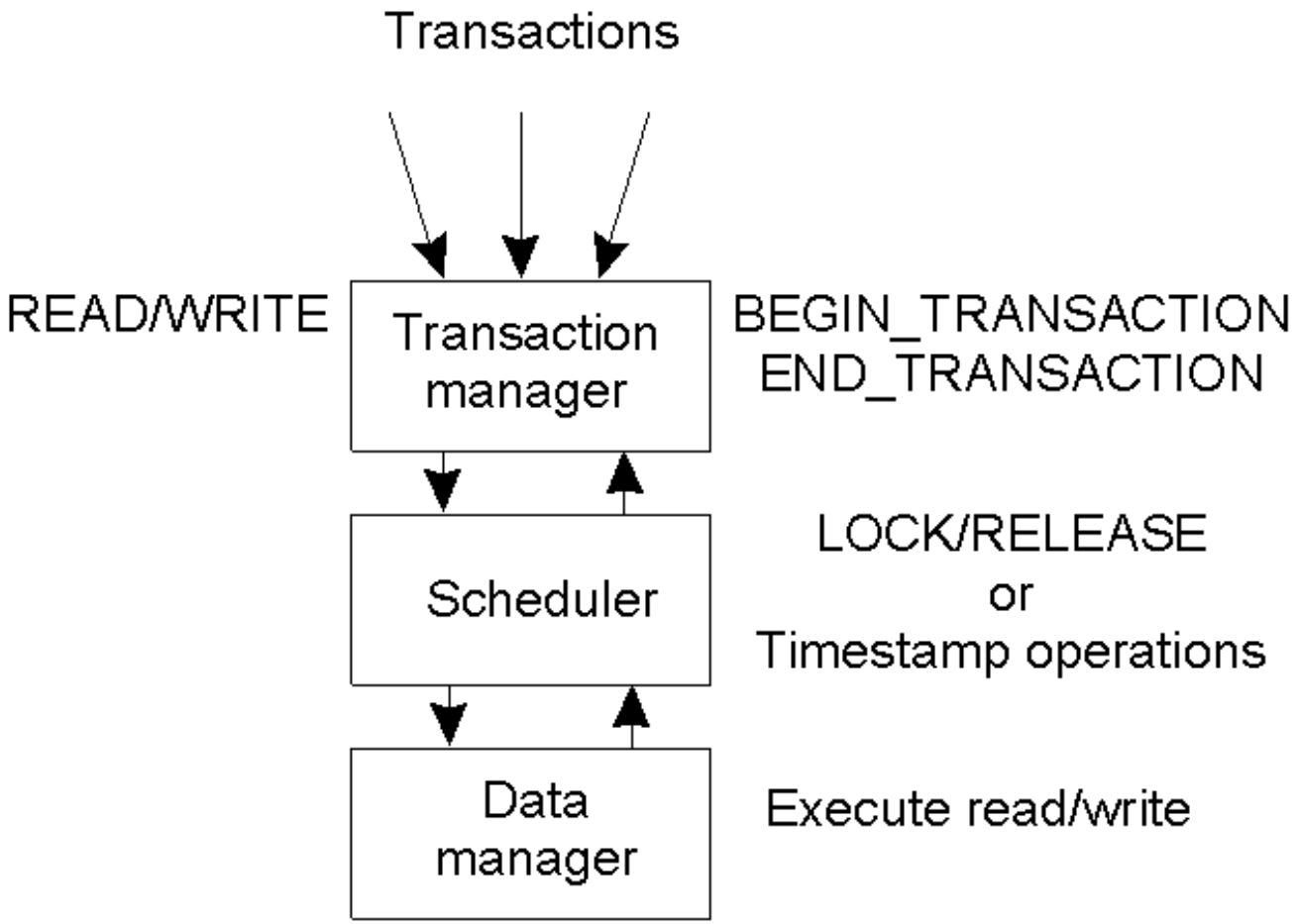
d) Possible schedules

# Read and write conflicts

| *Operations of different transactions* | | *Conflict* | *Reason* |
|---|---|---|---|
| *read* | *read* | No | Because the effect of a pair of *read* operations does not depend on the order in which they are executed |
| *read* | *write* | Yes | Because the effect of a *read* and a *write* operation depends on the order of their execution |
| *write* | *write* | Yes | Because the effect of a pair of *write* operations depends on the order of their execution |

- Conflicting operations
- a pair of operations conflicts if their combined effect depends on the order in which they were performed
  - e.g. read and write (whose effects are the result returned by read and the value set by write)

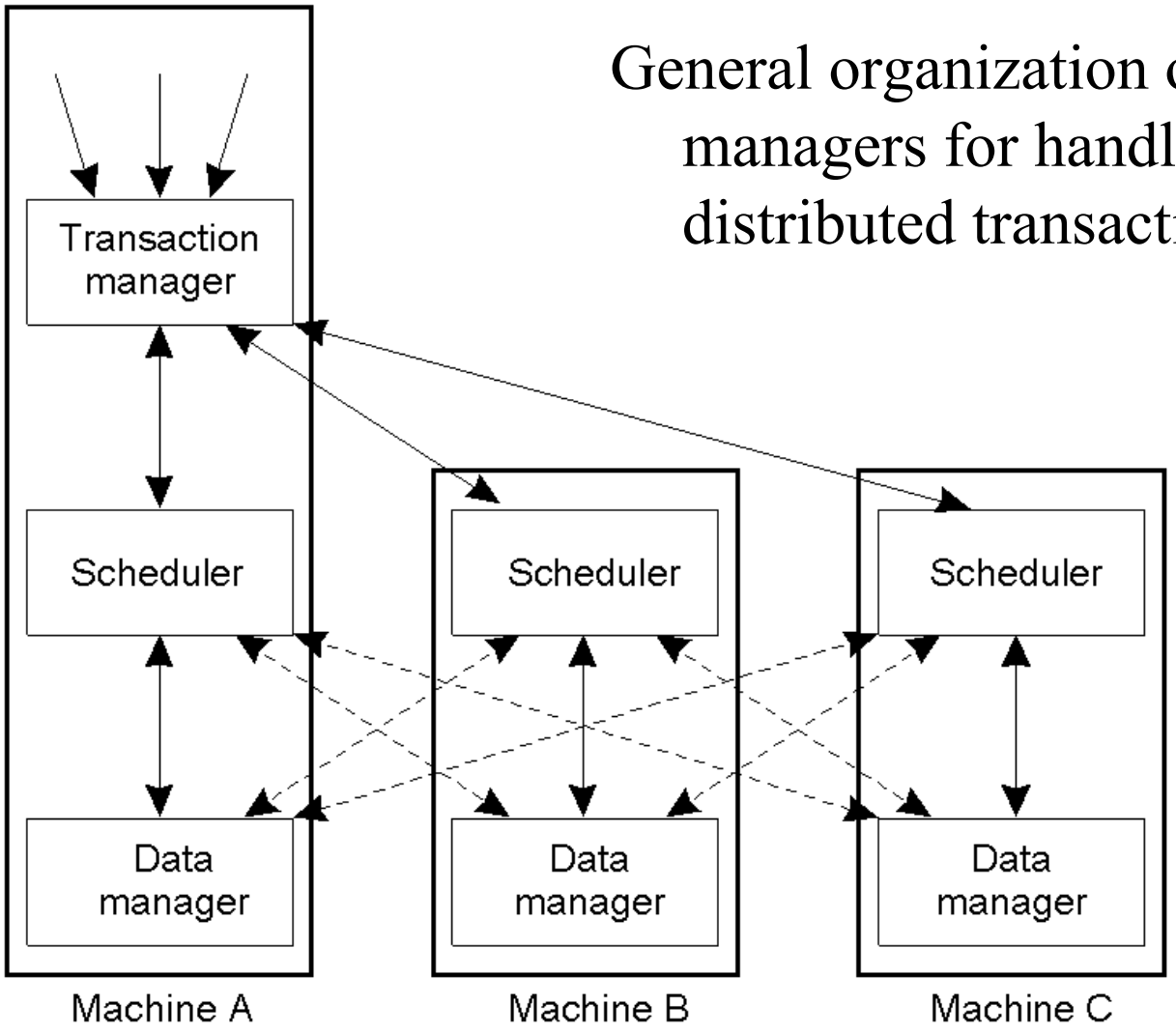
# Concurrency Control (1)



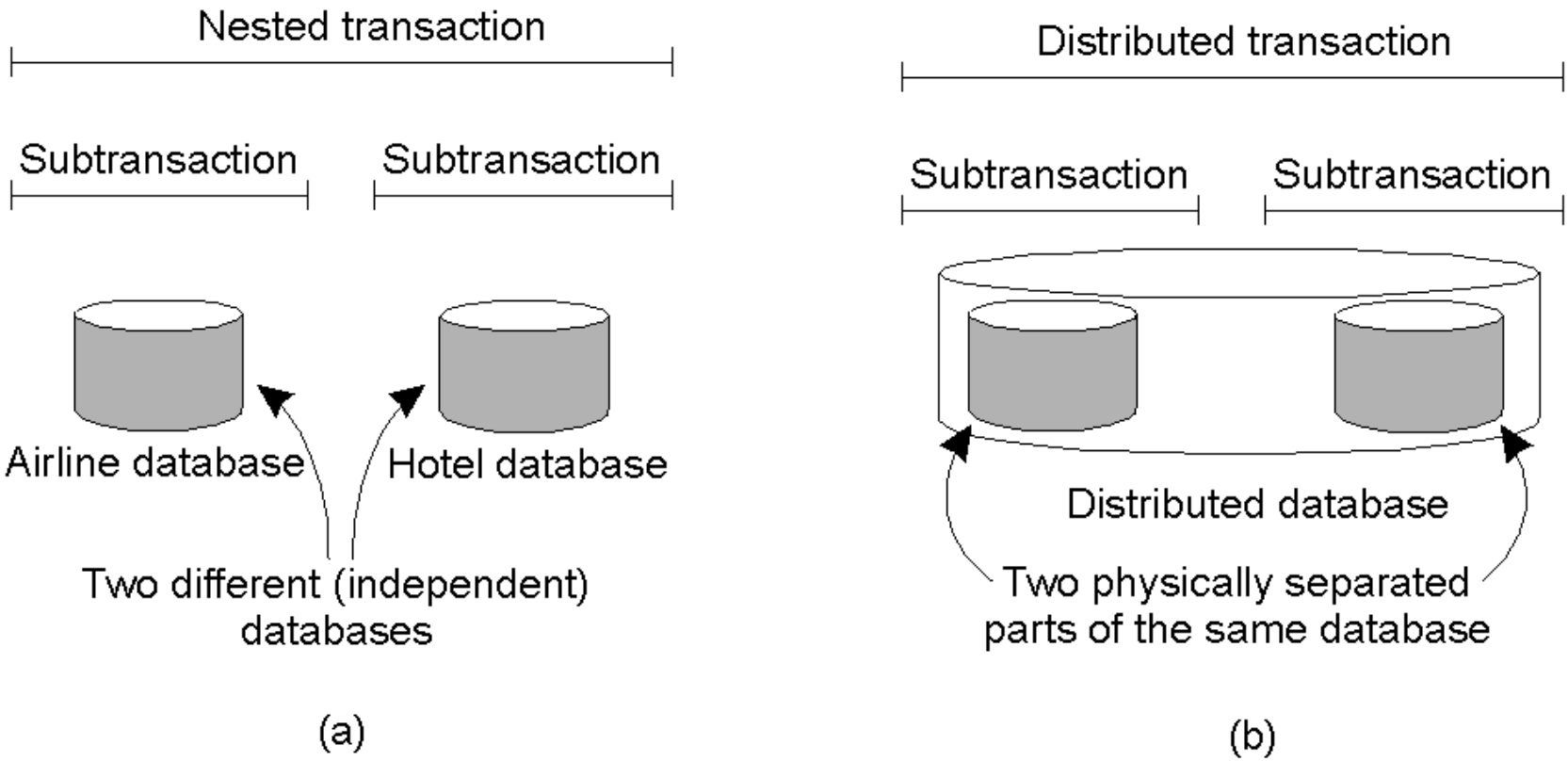

General organization of managers for handling transactions.

# Concurrency Control (2)

General organization of managers for handling distributed transactions.



# Nested vs. Distributed Transactions




a) A nested transaction – logically decomposed into a hierarchy of sub-transactions.

b) A distributed transaction – logically a flat, indivisible transaction that operates on distributed data.

# Implementation Issues

- How to undo actions
  - Private workspace
  - Writeahead log
- How to schedule actions
  - Lock-based approaches
    - Two-Phase Locking
  - Timestap-based approaches
    - Pessimistic
    - Optimistic
- Coordinating distributed transactions

# Private Workspace

- When a process starts a transaction it is given a private workspace containing all the files
- Until the transaction either commits or aborts, all of the reads and writes go to the private workspace

- Cost of copying everything!!!

# Writeahead Log (1)

- Files are modified in place, but a record is written to a log prior to that
  - Only changes the file after the log has been written successfully
- If the transaction aborts, the log can be used to **rollback** to the original state.

- Problem with concurrent reads and writes

# Writeahead Log (2)

```
x = 0;                              Log              Log              Log
y = 0;
BEGIN_TRANSACTION;
  x = x + 1;                        [x = 0 / 1]      [x = 0 / 1]      [x = 0 / 1]
  y = y + 2                                          [y = 0/2]        [y = 0/2]
  x = y * y;                                                          [x = 1/4]
END_TRANSACTION;
```

(a) (b) (c) (d)

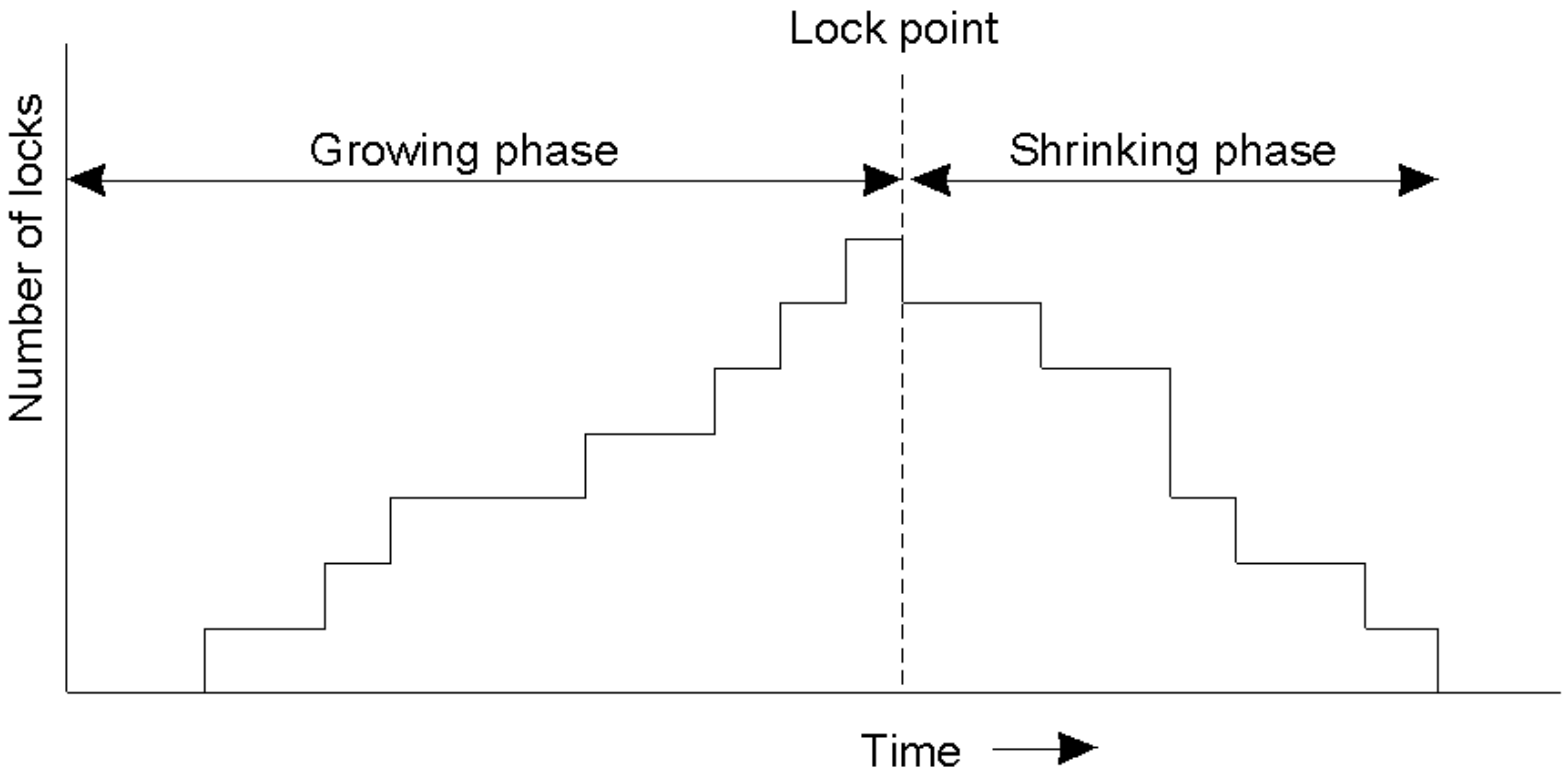
a) A transaction

b) – d) The log before each statement is executed

# Two-Phase Locking (1)

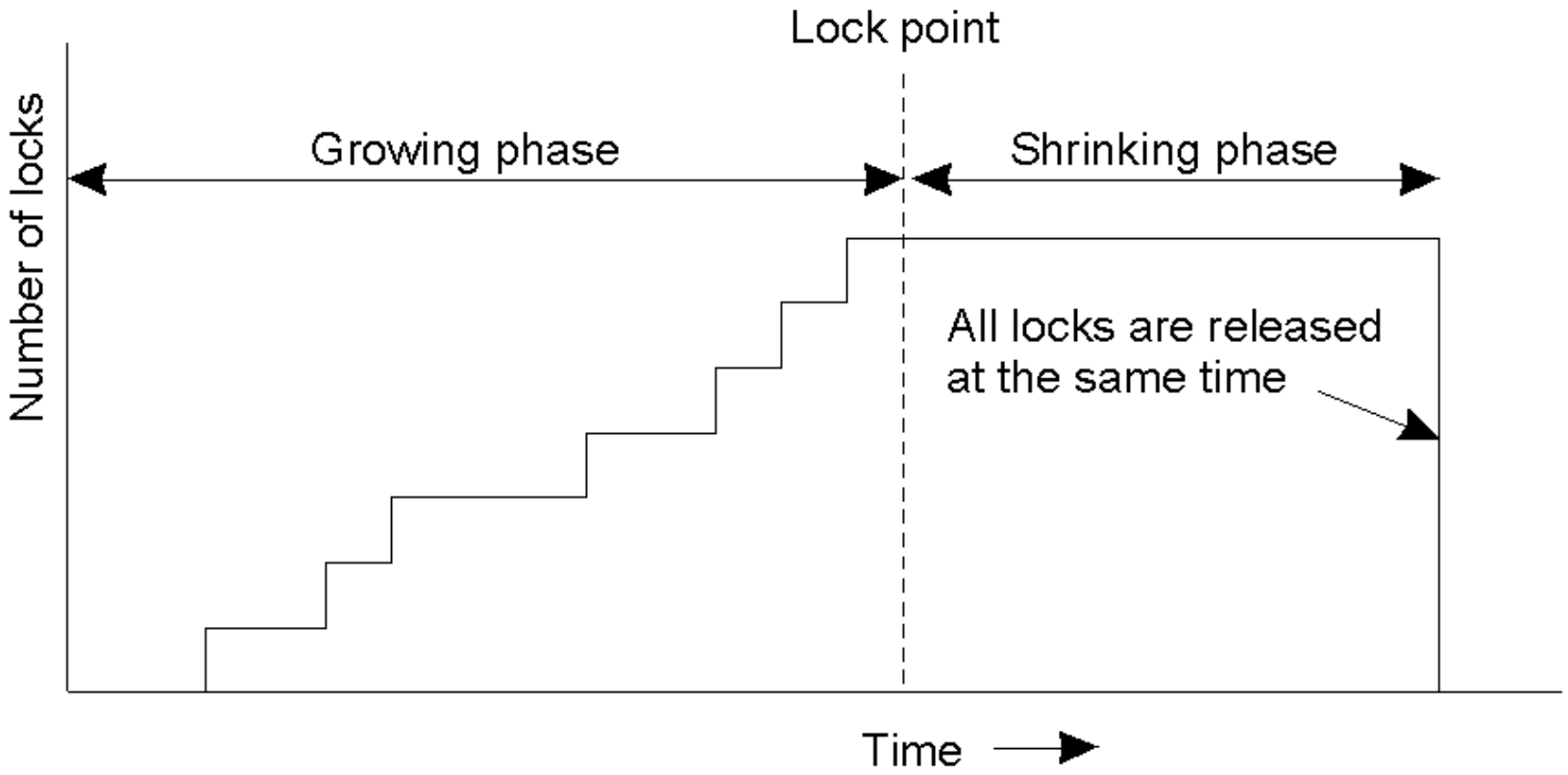
- Test whether operation *oper(T,x)* conflicts with any other operation for which a lock is already granted. If there is a conflict, delay *oper(T,x)*; otherwise, grant *T* a lock on *x*
- Never release the lock on *x* until the operation *oper* is performed
  - Once the lock is released, *T* cannot get it back

# Two-Phase Locking (2)




Two-phase locking **(2PL).**

# Two-Phase Locking (3)




Strict two-phase locking.

# Deadlocks

| Transaction *T* | | Transaction *U* | |
|---|---|---|---|
| **Operations** | **Locks** | **Operations** | **Locks** |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| • • • | waits for *U*'s | *a.withdraw(200);* | waits for *T*'s |
| | lock on *B* | • • • | lock on *A* |
| • • • | | • • • | |
| • • • | | • • • | |

**Deadlock** is a state in which each member of a group of transactions is waiting for some other member to release a lock.

# Dealing with deadlocks

- Acquire all locks in some predefined order
  - but this can result in premature locking and a reduction in concurrency
- Deadlock detection
  - after detecting a deadlock, a transaction must be selected to be aborted to break the cycle
  - it is hard to choose a "victim"
- Timeouts
  - if the system is overloaded, lock timeouts will happen more often and long transactions will be penalised
  - it is hard to select a suitable length for a timeout

# Pessimistic Timestamp Ordering (1)

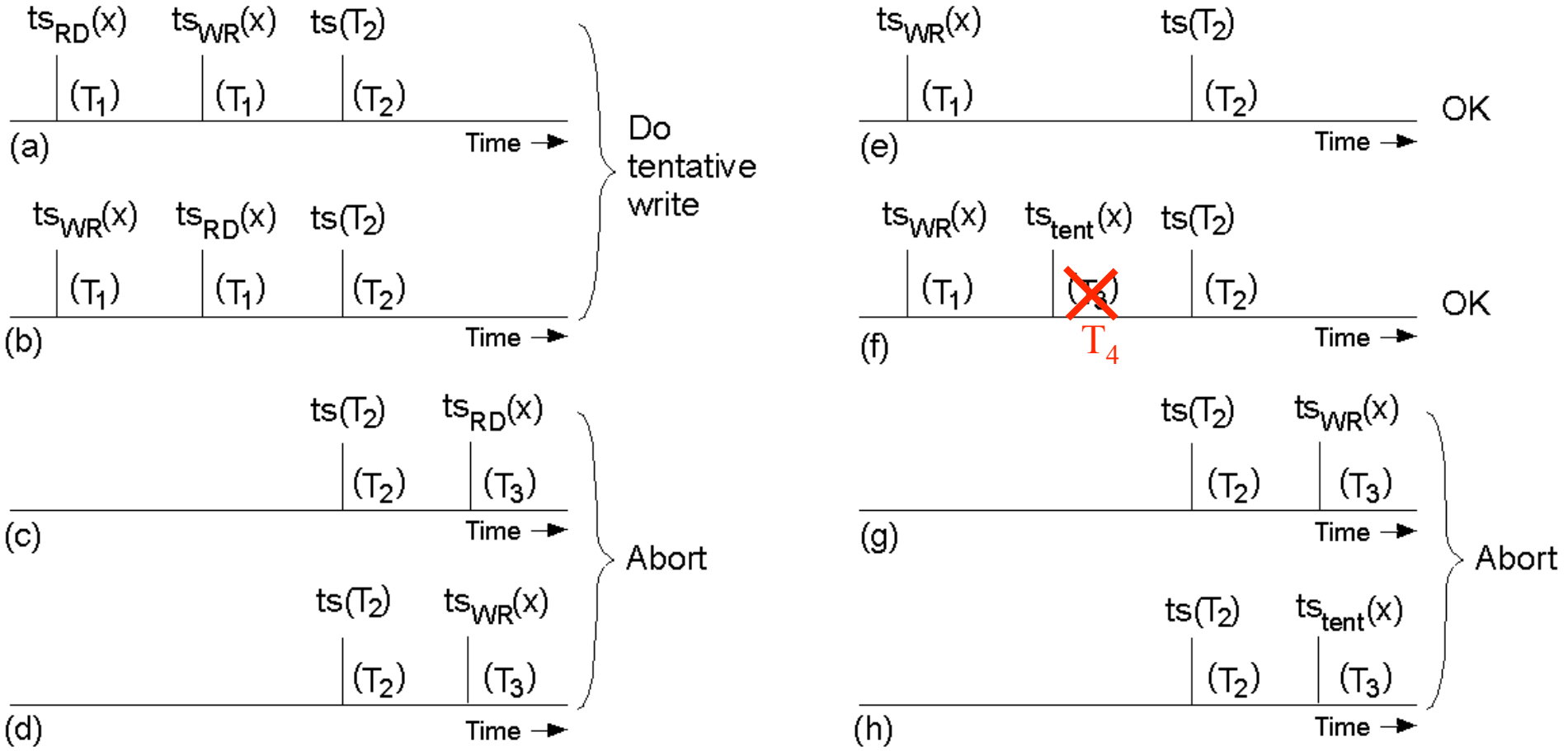
- Assign each transaction *T* a *unique* timestamp *ts(T)*.
- Every data item *x* has a read timestamp $ts_{RD}(x)$ and write timestamp $ts_{RD}(x)$ associated with them.
- A request to write an object is valid only if that object was last read and written by earlier transactions.
- A request to read an object is valid only if that object was last written by an earlier transaction.

# Example

Three transactions: $T_1$, $T_2$, $T_3$

- $T_1$ ran a long time ago, and committed
- $T_2$, $T_3$ are executed concurrently
- $ts(T_2) < ts(T_3)$

# Pessimistic Timestamp Ordering (2)




$T_2$ writes

$T_2$ reads

# Optimistic Concurrency Control

The scheme is called optimistic because the likelihood of two transactions conflicting is low

- A transaction proceeds without restriction until committing (no waiting, therefore no deadlock)
- It is then checked to see whether it has come into conflict with other transactions
- When a conflict arises, a transaction is aborted

# Comparison of Scheduling Methods

- Pessimistic approach (detect conflicts as they arise)
  - locking: serialisation order decided dynamically
  - timestamp ordering: serialisation order decided statically
  - timestamp ordering is better for transactions where reads >> writes,
  - locking is better for transactions where writes >> reads
  - strategy for aborts
    - timestamp ordering – immediate
    - locking– waits but can get deadlock
- Optimistic methods
  - all transactions proceed, but may need to abort at the end
  - efficient operations when there are few conflicts, but aborts lead to repeating work

# Distributed COMMIT

**General Goal:**

*We want an operation to be performed by all group members, or none at all.*

There are three types of a commit protocol: one-phase, two-phase and three-phase commit.

# Commit Protocols

- **One-Phase Commit Protocol**:
  - An elected co-ordinator tells all the other processes to perform the operation in question.

But, what if a process cannot perform the operation?
There's no way to tell the coordinator! Whoops …

- **The solutions**:

The *Two-Phase* and *Three-Phase Commit Protocols*.

# The Two-Phase Commit Protocol

First developed in 1978

*Summarized: GET READY, OK, GO AHEAD.*

1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. The group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator. A *GLOBAL_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL_ABORT* is sent.
4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

# Problem with Two-Phase Commit

- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.

- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers* ...

More of that when talking about fault tolerance